**Lecture T4: Pattern Matching**

---

**Introduction to Theoretical CS**

Two fundamental questions.
- What can a computer do?
- How fast can it do it?

General approach.
- Don’t talk about specific machines or problems.
- Consider minimal abstract machines.
- Consider general classes of related problems.

Today.
- Simplest type of machine that is still interesting = FSA.
- Class of (pattern matching) problems that it can solve.

Future lectures.
- More complicated machines and problems.

---

**Why Learn Theory**

In theory . . .
- Deeper understanding of what is a computer and computing.
- Foundation of all modern computers.
- Pure science.
- Philosophical implications.

In practice . . .
- Web search: theory of pattern matching.
- Sequential circuit: theory of finite state automata.
- Compilers: theory of context free grammar.
- Cryptography: theory of complexity.

---

**Web Search Example**

Standard Web search for ‘+censorship +net’ might yield 1 million hits, ordered as follows:
- [www.epic.org/free_speech/action](http://www.epic.org/free_speech/action)
- [www.tiac.net/users/sojum/censor0596.html](http://www.tiac.net/users/sojum/censor0596.html)

Observation: not many useful pages.
Web Search Example

Standard Web search for `+censorship +net` might yield 1 million hits, ordered as follows:

- Jon Kleinberg’s clever algorithm produces “authoritative” pages without human fine-tuning:
  - www.eff.org (Electronic Frontier Foundation)
  - www.cdt.org (Center for Democracy and Technology)
  - www.vtw.org (Voters Telecommunications Watch)
  - www.aclu.org (American Civil Liberties Union)

- Based on theoretical principles.
- Involves computing eigenvectors of Web matrix!
- Use google.com

Unix Tools

- A large number of simple tools.

Some fundamental pattern matching tools.
- egrep, awk, sed, more, emacs, perl
- Useful for variety of applications including Web search.
- Not C programming, though tools are as powerful.
- Directly related to fundamental tenets of computer science.

egrep

General regular expressions pattern matching.

- Acts as filter.
- Sends lines from stdin to stdout that “match” argument string.

**Elementary Examples**

```bash
% egrep 'beth' classlist
 02/Condliffe/Elizabeth/3/condlife
 03/Danaher/Elizabeth/8/edanaher
 03/Smythe/Elizabeth/6/esmythe
 03/Bethke/Kristen/3/kbethke

% egrep '/3/' classlist
 03/Marin/Anthony/3/amarin
 03/Parker/Andrew/3/aparker

% egrep 'zeuglodon' mobydick.txt
rechristened the monster zeuglodon and in his
```

Crossword Puzzle or Scrabble Too Hard?

/usr/dict/words is a list of (25,143) words in dictionary.

**More Examples**

```bash
% egrep 'hh' /usr/dict/words
beachhead
highhanded
withheld
withhold

% egrep 'u.u.u' /usr/dict/words
cumulus

% egrep 'oo..oo' /usr/dict/words
bloodroot
schoolbook
schoolroom

% egrep -c 'oo..oo' /usr/dict/words
3
```

- A dot matches any single character
- Why not “cookbook”?
- Two consecutive h’s
- A dot matches any single character
- count number of matches
Excerpts From "man egrep"

Unix

% man egrep
Name: egrep - search file using full regular expressions
Syntax: egrep [option...] expression [file...]
Description: Search the input files (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output.
Take care when using special characters in the expression because they are also meaningful to the Shell. It is safest to enclose the entire expression argument in single quotes.
Options:
- c Produces count of matching lines only.
- i Considers upper and lowercase letter identical.
- n Precedes each matching line with its line number.
- v Displays all lines that do not match.
Restrictions: Lines are limited to 256 chars; longer lines are truncated.

Grep Pattern Conventions

Conventions for egrep:
- c any non-special character matches itself
- . any single character
- r* zero or more occurrence of r
- {r} grouping
- r1|r2 logical OR
- [...] any character in [a-z]
-[^ ...] any character not in [a-z]
- ^ beginning of line
- $ end of line

Still More Examples

Unix

% grep 'n(ie|ei)ther' /usr/dict/words
neither
% grep 'actg(atac)*gcta' human.data
ggt actg gcta ggac
% grep 'actg(atac)*gcta' student.data
tat actg atacatacatac gcta ttac
% grep '^y.(..)*y$' /usr/dict/words
yesterday
% grep -v '[aeiou]' /usr/dict/words | grep '.......
  rhythm
  syzygy

Pattern Matching Alternatives in Unix

Pattern matching.
- grep, egrep
- more

Substitution editing.
- emacs, ex
- sed: filter, line by line substitution
  sed 's/apples/oranges/g' file.txt

Pattern matching languages.
- awk, perl
- Matching, substitution, pattern manipulation, variables, numeric capabilities, control and logic.
### Regular Expressions

Specifying "pattern" for grep can be complex.

```
[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*
```

What kinds of patterns can be specified?
- Match all lines containing an even number?
- Match all lines containing a prime number?

Which aspects are essential?
- Unix `egrep` regular expressions are useful.
- But more complex than theoretical minimum.

### Rules for creating regular expressions (RE's):

- **Symbols**: 0 or 1
- **Grouping**: a
- **Concatenation**: ab
- **Logical OR**: a + b
- **Closure**: a*

where a and b are regular expressions.

**Examples:**

- `(10)*`: ε, 10, 1010, 101010, ...
- `(0 + 1)*0`: 00, 000, 010, 0000, 0110, ...
- `(1*01*01*01*)*`: ε, 000, 000000, 11101110101111, ...

### Formal Languages

An alphabet is a finite set of symbols.
- Binary alphabet = \{0, 1\}
- Lower-case alphabet = \{a, b, c, d, ..., y, z\}
- Genetic alphabet = \{a, c, t, g\}

A string is a finite sequence of symbols in the alphabet.
- '0111011011' is a string in the binary alphabet.
- 'tigers' is a string in the lower-case alphabet.
- 'acctgaacta' is a string in the genetic alphabet.

A formal language is an (unordered) set of strings in an alphabet.
- Can have infinitely many strings.
- Examples:
  - \{0, 010, 0110, 01110, 011110, ...\}
  - \{11, 1111, 111111, 11111111, ...\}

### Why Study Formal Languages?

Can cast any computation as a language recognition problem.
- Is x = 23,536,481,273 a prime number?
Regular Languages

Every RE describes a language (the set of all strings that match).
- \((1^*01^*01^*)^*\) describes the language of all bit strings with an even number of 0’s:
  \{1, 11, 1001, 0010110111, ...\}

Regular language.
- Any language that can be described by a RE.

Which languages are regular? (all but one of the following)

<table>
<thead>
<tr>
<th>All bit strings that</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin with 0 and end with 1.</td>
<td>0010101011</td>
</tr>
<tr>
<td>Have a multiple of 3 0’s.</td>
<td>11000110100</td>
</tr>
<tr>
<td>Have more 1’s than 0’s.</td>
<td>01111001100</td>
</tr>
<tr>
<td>Have no consecutive 1’s.</td>
<td>01001010010</td>
</tr>
</tbody>
</table>

Machines

Can cast any computation as a language recognition problem.
- Is \(x = 23,536,481,273\) a prime number?
  - \(L = \{2, 3, 5, 7, 11, 13, 17, ...\}\)
  - Is \(x\) in language \(L\)?

Start by trying to understand simple languages.
- Build a machine to recognize regular languages.

Finite State Automata

Simple machine with \(N\) states.
- Start in state 0.
- Read an input bit.
- Move to new state
  - depends on input bit and current state
- Stop when last bit read.
  - ‘yes’ if end in accept state(s)
  - ‘no’ otherwise

‘Yes’ also called accepted or recognized inputs from a language.
- FSA to right accepts all bit strings of the form \(10(10)^*\)
- Rejects all others.

C Code for \(10(10)^*\) FSA

```c
int main(void) {
  int c, state = START_STATE;
  while ((c = getchar()) != EOF) {
    if (state == 0 && c == '0') state = 2;
    if (state == 0 && c == '1') state = 1;
    if (state == 1 && c == '0') state = 3;
    if (state == 1 && c == '1') state = 2;
    if (state == 2 && c == '0') state = 2;
    if (state == 2 && c == '1') state = 2;
    if (state == 3 && c == '0') state = 2;
    if (state == 3 && c == '1') state = 1;
  }
  if (state == ACCEPT_STATE)
    printf("Accepted\n");
  else
    printf("Rejected\n");
  return 0;
}
```

fsa1.c
Better C Code for FSA

```c
#include <stdio.h>
#define STATES 4
#define ALPHABET_SIZE 2
#define START_STATE 0
#define ACCEPT_STATE 3

int main(void) {
    int c, state = START_STATE
    int transition[STATES][ALPHABET_SIZE] =
        { {2, 1}, {3, 2}, {2, 2}, {2, 1} };

    while ((c = getchar()) != EOF)
        if (c >= '0' && c < '0' + ALPHABET_SIZE)
            state = transition[state][c - '0'];
        else if (state == ACCEPT_STATE) printf("Accepted\n");
        else printf("Rejected\n");
    return 0;
}
```

A Second Example

Consider the following two state FSA.

What bit strings does it accept?
- Yes: 0, 11110, 00000, 10010111011, all bit strings with an odd number of 0's.
- No: 1, 1111, 00, 1011100111011, all bit strings with an even number of 0's.
- Recognizes same language as \((1*01*01*)^* (1*01*)^*\)

A Third Example

Build an FSA that accepts all strings that contain 'acat' as a substring.
- tgacatg
- acacatg

Finish building:

More Applications
Duality Between FSA's and RE's

Observation: every FSA we create generates a regular language.
(the inputs accepted can be specified by a regular expression)
Is this always the case?

What about the OTHER way around? Suppose I give you a regular expression. Can you always create a FSA that accepts precisely the same strings?

Yes. It's no coincidence.

What about the OTHER way around? Suppose I give you a regular expression. Can you always create a FSA that accepts precisely the same strings?

Yes.

I don't see why?

Stay tuned: see Lecture T2.

Limitations of FSA

No FSA can recognize the language of all bit strings with an equal number of 0's and 1's.

Suppose an N-state FSA can recognize this language.
Consider following input: 00000001111111, N+1 0's N+1 1's

FSA must accept this string.
Some state x is revisited during first N+1 0's since only N states.

Machine would accept same string without intervening 0's.

This string doesn't have an equal number of 0's and 1's.

Limitations of Regular Languages

Consequence: there are languages that are not regular.
No FSA can recognize the language of all bit strings with an equal number of 0's and 1's.
We claimed that FSA's are equivalent to regular expressions, i.e.,
for any regular language, there is a FSA that accepts precisely those strings
the language of all strings accepted by any specific FSA is regular.
Hence, the language above cannot be regular.
Looking Ahead

Today.
- Defined a simple abstract machine – FSA.
- Capable of pattern matching.
- Incapable of “counting.”
- Need to consider more powerful machines.

Future lectures.
- Define an abstract machine.
- Understand how it works and what it can do.
- Find things it can’t do.
- Define a more powerful machine.
- Repeat until we run out of problems or machines.

A Fourth Example

FSA to decide if input (convert binary to decimal) is divisible by 3?

How does it work?
- State 0: input so far is divisible by 3.
- State 1: input has remainder 1 upon division by 3.
- State 2: input has remainder 2 upon division by 3.
- Transition example.
  - Input 1100 (12) ends in state 0.
  - If next bit is 0 then stay in state 0. Adding 0 to last bit is same as multiplying number by 2. Remains divisible by 3.

An Application: Bounce Filter

Bounce filter: remove isolated 0’s and 1’s in input.
- Input: 0 1 0 0 0 1 1 0 1 1 1 1 1
- Output (one-bit delay): 0 0 0 0 0 1 1 1 1 1 1 1 1
- x/y: if input is x, then change state and OUTPUT y

Hmm. Which will we run out of first?
An Application: Bounce Filter

Bounce filter: remove isolated 0's and 1's in input.
- Input: 0 1 0 0 0 1 1 0 1 1 1 1
- Output (one-bit delay): - 0 0 0 0 0 1 1 1 1 1 1 1
- \( x/y \): If input is \( x \), then change state and OUTPUT \( y \)

State interpretations.
- 0: at least two consecutive 0's
- 1: sequence of 0's followed by a 1
- 2: at least two consecutive 1's
- 3: sequence of 1's followed by a 0